Modiator - A Web App for Modelica Simulation

Hilding Elmqvist[1] Martin Otter[2]

[1] Mogram AB, Sweden, hilding.elmqvist@mogram.net

[2] DLR, Institute of Vehicle Concepts, Germany, martin.otter@dlr.de

Abstract

The Modelica language (www.modelica.org) has become a de facto standard for systems modeling and many tools exist. This paper describes certain modern enhancements and a static web app implementation called *Modiator* (*Modelica Instant Simulator*). It allows an improved immediate first-time user experience since the web app is available in seconds and simulations can be done directly in the browser. State of the art numerical solvers from the Sundials suite have been compiled into WebAssembly. The Modelica model is translated into Javascript code using techniques such as sorting, tearing, index reduction, state selection, etc. A subset of Modelica is supported with some extensions, for example, support for self-modifying models. This paper also presents the Stream and Model3D prototype libraries.

Keywords: Modelica, Modia, Modia3D, compilation, web app, WebAssembly, simulation in the browser, cloud computing, serverless functions, Monte Carlo simulation

1 Introduction

The Modelica language (Modelica Association 2023) makes it easy to build large, complex models since instantiation of reusable component models is possible. However, current tools[1] usually make full expansion of the equation sets including scalarization of arrays for all instances and transform the equations to C-Code that is compiled and linked. This means that translation time can be minutes. There is also a need for self-modifying models for example to support adaptable grid for finite volume models, variable model structures such as multi-stage rockets or just turning on and off subsystems. This means that separate translation would be beneficial, i.e. only translating the parts of the model that has been changed. This will also speed up experiments involving variant selection by simulation when certain subsystems are redeclared to analyze alternative designs.

Web technology allows an improved immediate first-time user experience since the web app needs no installation (and also no administrator rights), is available in seconds and faster separate translation and simulations done directly in the browser means less waiting. This will also make object-oriented modeling and simulation more accessible to students, hopefully increasing the popularity of Modelica.

Mobile phones are becoming more popular than computers and they have sufficient computing power for simulation. The drawback of mobile phones is the limited screen size (except for foldable phones). One possible solution is to use an *infinite canvas* and convenient and fast zooming, scrolling, and panning.

For multi-simulations for variant selection or Monte Carlo simulations, cloud should be possible to utilize, for example by serverless lambda functions, and there are needs to visualize results in different ways than just trend plots, such as scatter diagrams and histograms.

Many of the innovations and findings in this paper regarding translating models come from the experiences from the Modia project[2], see, e.g., (Elmqvist et al. 2021). Modia is a language with many similarities to Modelica. However, the language is much simpler than the Modelica language, and implementation of the Modia environment is based on Julia[3] (Bezanson et al. 2017). The effort is now moved from the Modia language to the Modelica language for three reasons: (1) It takes too long to get the first plot when using Modia (installing Julia and many used packages, as well as compiling generated Julia model code the first time). (2) There are many existing Modelica model libraries in wide-spread use and it is too much effort to develop and maintain a thorough translator from Modelica to Modia. (3) New technologies with WebAssembly [4] and fast just-in-time compilation of Javascript enables immediate and fast simulations.

The paper introduces *Modiator* (*Modelica Instant Simulator*), a static web-app implemented with Javascript and WebAssembly providing support for a sub-set of the Modelica language with some extensions of Modelica based on experience with Modia/Julia and taking advantage of modern web technology.

The commercial Modelica environment *Modelon Impact*[5] *(Elmqvist et al. 2018)* also provides a browser GUI. However, compilation and simulation of models are not performed in the browser but via a cloud service that has to be paid for. The default behavior of Modiator is that models are compiled and simulated locally in the browser.

[1] https://modelica.org/tools/

[2] https://github.com/ModiaSim/Modia.jl

[3] https://julialang.org/

[4] https://webassembly.org/

[5] https://modelon.com/modelon-impact/

This requires different techniques, e.g., to generate model code directly in Javascript, instead of in C or C++ (as done by other Modelica tools).

Compiling and simulating a Modelica model in a web browser has been proposed by (Franke 2014) and demonstrated with a tiny, very partial prototype. (Short 2014) demonstrates a tool chain to translate Modelica models to Javascript based on the OpenModelica [6] compiler and Emscripten[7] and run the Javascript code locally in the browser. (Kulhánek et al. 2023) provide the Bodylight.js 2.0 toolchain to build in-browser web simulators by translating a Modelica model into an FMU[8] (Functional Mockup Unit), use Emscripten to transform the C-source code inside the FMU into WebAssembly format and embed this code in a web page with a specific user-friendly GUI for this particular Modelica model.

2 A Modernized Modelica Syntax

There are trends concerning the syntax of new programming languages (e.g. C#, Go, Julia, Python, Swift) which have been adopted by Modia. The following proposals for the Modelica language are inspired by Modia, based on these new trends:

- Using Greek letters in identifiers is natural for users making mathematical models.
- Semicolon is becoming optional which makes models easier to read. Traditionally, semicolons were used to enable resynchronization after the first error in order to continue parsing. The speed of parsing is now much faster, so it is better to abort after the first error is found.
- Type and unit inference and checking can be introduced if a notation is utilized to associate units to numeric literals. In the example below, units are appended to parameters and start values enclosed in single quotes. Most type specifiers are omitted since all parameters are Real and the states must be Real.
- The class identifier after end is made optional.

Below is shown a second order model utilizing the simplified, more readable syntax in Modiator:

Listing 1. Modelica model with the proposed condensed syntax and Greek identifiers.

```
model SecondOrder
  τ = 1.0 's'  // instead of: parameter Real tau(unit=”s”) = 1.0;
  ζ = 0.5
  input u
  x1(start=0)
  x2(start=0 '1/s')   // instead of: Real x2(start=0, unit=”1/s”);
  output y=x1
equation
  der(x1) = x2
  τ*τ*der(x2) + 2*ζ*τ*x2 + x1 = u
end   // instead of: end SecondOrder;
```

3 Self-Modifying Models

A much-wanted Modelica feature is to be able to change array dimensions without recompiling when, for example, higher fidelity of discretization is needed. Other needs are to be able to redeclare component classes or start and stop subsystems. These scenarios can be achieved by scripting repeated simulations where the start value for the next simulation is set to the end value of the previous simulation. At certain conditions, parameters can be changed, redeclarations made and some transformation algorithm is applied between final state and new state. Writing such scripts is not object oriented though. To make it reusable, the transformations should be made in the model classes.

A similar situation exists when updating HTML pages. The HTML DOM (Document Object Model) is a tree of various kinds of nodes which can be modified by Javascript functions which are called when different kinds of events occur. The DOM tree can also be traversed and a node id can be search for. Nodes can then be modified either by changing attributes or making a new subtree by providing an *innerText* definition. Nodes can also be added or removed by built-in JavaScript function calls.

The ModelManagement.Structure.AST functions of Dymola[9] can manipulate the AST (Abstract Syntax Tree) from Modelica scripts (not from Modelica models). This functionality can serve as inspiration for models manipulating themselves.

For this feature, Modelica needs to have an event detection mechanism (c.f. addEventListener in HTML). Modelica already has a when-statement which can be generalized. The model below shows some of the functions which can be used to manipulate the **ast** (reserved Modiator identifier for the Abstract Syntax Tree) of a model.

Listing 2. Model that modifies its definition during simulation.

```
model TestSelfModifications
  parameter Real T=1;
  Real x(start=0);
equation
  T*der(x) + x = 1;
  when x>0.3 then
    ast.setParameter('T', 0.1);
  end when;
  when time>0.5 then
    ast.setStart('x', 2);
  end when;
  when time>0.7 then
    ast.addEquation('0.1*der(y) + y = 1');
  end when;
end TestSelfModifications;
```

If a significant event occurs, the modified model ast is translated to Javascript code which is just-in-time compiled before the simulation continues.

[6] https://openmodelica.org/
[7] https://emscripten.org/
[8] https://fmi-standard.org/
[9] https://www.3ds.com/products/catia/dymola

DOI
10.3384/ecp218211

The semantics of the above function calls are explained by the plot below.

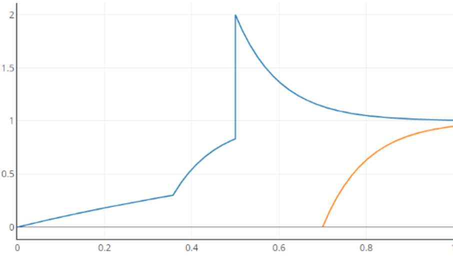


Figure 1. Plot of model TestSelfModifications

Note, that when the equation was added, also a new variable was introduced, y. By default, all state variables are currently plotted, i.e. y was also plotted during its existence. Adopted from Modia, Modiator does not require declaration of variables and the start value is 0 by default.

Array dimensions and the number of states can also be changed at events which is illustrated by the following model:

Listing 3. Model that modifies array dimensions and number of states during simulation.

```
model VaryingDimensions
  Integer n=2;
  Real x(start=1:n, fixed=true);
equation
  der(x) = -diagonal(1:n)*x;
  when time > 0.1 then
    ast.setParameter('n', 3);
    ast.setStart('x[3]', (x[1]+x[2])/2);
  end when;
  when time > 0.2 then
    ast.setParameter('n', 2);
    ast.setStart('x[1]', (x[1]+x[3])/2);
    ast.setStart('x[2]', (x[2]+x[3])/2);
  end when;
end VaryingDimensions;
```

Note, the new start values defined with ast.setStart(…) are calculated before the model is evaluated at the event. This gives a nice mechanism for transferring the state information to a new state representation. See the results below.

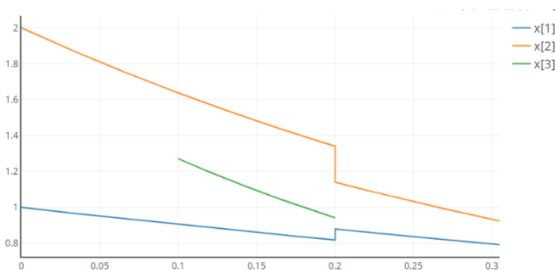


Figure 2. Plot of model VaryingDimensions.

In the companion paper on resizable arrays (Otter et al. 2025), this technique is used for adaptive grid modeling of an insulated rod.

4 Web App for Immediate Results

Web technology allows an immediate first-time user experience since a web app is available in seconds and simulations can be done directly in the browser.

Mobile phones are becoming more popular and powerful than laptop computers and they have sufficient computing power for simulation. The drawback is the limited screen size. One possible solution is to use an *infinite canvas* (inspired by Apple Photos) and convenient and fast zooming, scrolling, and panning. The DOM/CSS-feature *style.transform* [10] provides the fundamental rendering functionality to implement infinite canvas in the browser. Modiator uses two-finger touches on touch screens to zoom and move around.

The Modiator User Interface is based on a set of widgets (text editor, plots, 3D animator, model diagrams, parameter dialog, etc.) which can be freely placed and sized on the *infinite model board*, see Figure 3:

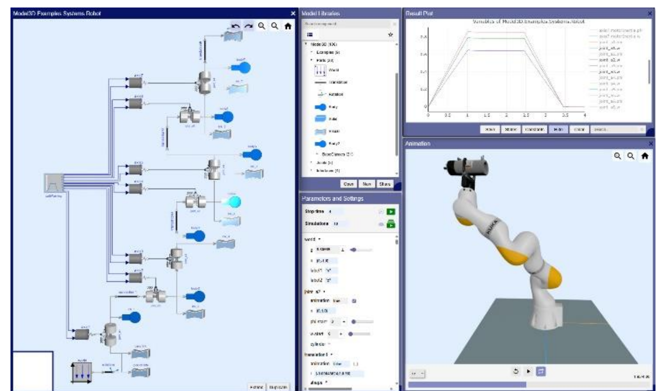


Figure 3. Example of Modiators User Interface.

The model diagram rendering is inspired by the 2.5D look of Playmola (Elmqvist et al. 2015), i.e., that icons have a 3D representation which are placed in a 2D diagram.

Multiple plot widgets can be used to visualize simulation results in different ways such as line plots, scatter diagrams, etc.

To enable AI techniques for Modelica authoring, external text editors such as VS Code with Github Copilot plug-in, can be used. When the external file is changed, Modiator will load and simulate the updated model.

The Modiator web app is a PWA (Progressive Web App), i.e., it can be installed on your device and by using service workers it can be used offline without internet connection.

[10] https://www.w3schools.com/jsref/-prop_style_transform.asp

5 Simulation in the Browser

Modiator uses a mixed execution model for simulation. Modelica models are translated to JavaScript functions which calculate the derivatives given time, states and parameters. The integration of the differential equations is performed by CVODE from the SUNDIALS suite [11] (Hindmarsh et al. 2005, Gardner et al. 2022). CVODE has been translated from C to WebAssembly [12] utilizing Emscripten [13] and is called from Javascript when simulation starts. When derivatives need to be calculated, a Javascript callback function is called from CVODE. Special considerations had to be taken especially regarding allocation and transfer of arrays.

Modiator translates the model equations to a JavaScript function by using the *new Function()* concept[14]. The Function constructor takes two arguments. The first one is a string of comma-separated arguments of the function and the second is the body of the function as a string, for example:

```
const func = new Function('a,b', 'return a + b')
console.log(func(1, 2)) // 3
```

Such functions are usually Just-In-Time-compiled. Array and matrix equations are not expanded (Otter et al. 2025), but evaluated by calls to run-time functions and math.js[15].

In order not to disturb user interactions, parsing, translation and simulations are performed in concurrent web workers.

6 Simulation on the Cloud

For multi-simulations for variant selection or Monte Carlo simulations, it should be possible to utilize cloud computations. So far, *Netlify functions* [16] have been utilized. They are based on Amazon Web Services Lambdas.

Essentially the same simulation code as for web workers can be used. Instead of posting simulation data to a web worker, *fetch* requests are made to a simulate API function with a *body* of simulation data. The simulation data contains parameters and the JavaScript function string for calculating derivatives. To support multi-simulations with redeclares or self-modifying models, the Modelica translator module is also available in the serverless functions.

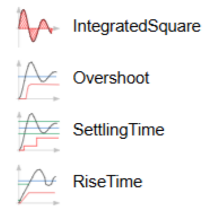
So far, limited testing has been made. 10000 simulations have been performed on the cloud. The startup time for each simulation was less than a second.

7 Monte Carlo Simulation

Model-based product design involves determining product topology, component selection, component sizing, and parameters (tuners) in such a way that the product has good performance, low production cost and low cost of ownership (objectives) while being robust with regards to variations in its environment and insensitive to variations in parameters (uncertainties). Since topology, component selection and component sizing are important, gradient based optimization tools can't be used.

Modiator uses Monte Carlo simulation and randomly selects tuner values typically from uniform distributions and selects uncertainties typically from truncated normal distributions. Redeclarations to model design selections or discrete environment variations have associated discrete distributions.

A set of Modelica *Objective* blocks is available for inserting into a model to define what are good designs. A subset of these blocks is shown on the right.



Since several possibly conflicting objectives can be defined, the *Pareto frontier* in scatter diagrams can help with compromise decisions and the *parallel axis coordinates plot* can help with assigning different weights to the objectives.

Equality and inequality constraints are enforced by including assert statements that cause simulations to fail if the specified conditions are not met. Functions of the following kind are provided for defining tuners and uncertain parameters:

```
p=uniform(default, minimum, maximum, tuner)
```

Such expressions can be used in modifiers. The *default* value is used for tools that do not provide Monte Carlo simulation in this way. *Minimum* and *maximum* gives the range and *tuner* is true for a tuner, otherwise an uncertainty is defined. The user interface has built-in menus for defining such function invocations (± button).

Figure 4. Tuner and Uncertainty definitions.

[11] https://webassembly.org/
[12] https://webassembly.org/
[13] https://emscripten.org/
[14] https://javascript.info/new-function
[15] https://mathjs.org/
[16] https://www.netlify.com/platform/core/functions/

DOI 10.3384/ecp218211

7.1 Design Example – Servo

To illustrate the design method used by Modiator, a well-known control problem will be employed: PID control of a double integrator. The motor consists of an ideal torque generator and an inertia and behaves like a double integrator.

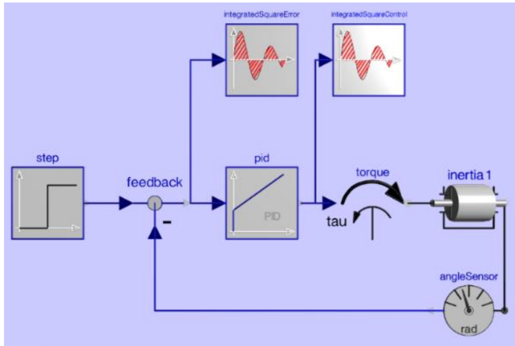


Figure 5. Controlled double integrator.

Two integral square objectives are defined. In addition to the usual objective of small control error, it is desired to keep the control effort small to save energy. The gain, k, and derivative time constant, Td, should be tuned, i.e. uniform distributions are defined and tuner is set to true for them:

```
pid(k=uniform(1, 0.1, 1, true), Td=uniform(2, 0.5, 5, true))
```

By running 100 simulations, the following spaghetti plot is obtained if angleSensor.phi and step.y were plotted during a previous single simulation:

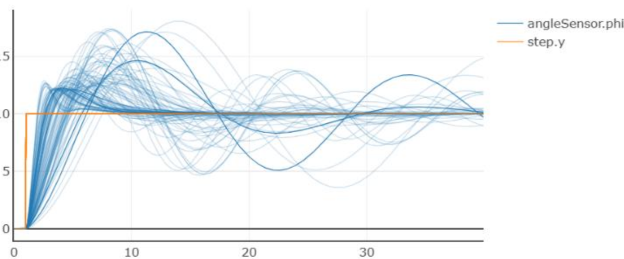


Figure 6. Spagetti plot for 100 simulations.

The time to make one thousand simulations is 2.24 seconds on a HP ZBook Ultra G1a laptop with an AMD RYZEN AI MAX+ PRO 95 3.0 GHZ CPU with 16 cores/32 threads. The corresponding time for MacBook Pro M4 is 1.56 seconds and for iPad Pro 11” M4 is 2.71 seconds. Amazingly, the same task takes only 4.32 seconds on a Samsung Galaxy Z Fold6 mobile phone.

In addition, Modiator presents a matrix of plots, see **Figure 7**. The rows and columns represent the variables: integratedSquareError.y, integratedSquareControl.y, pid.k and pid.Td. The diagonal shows the histograms for the variables (objectives: blue, tuners: green. Plot 1,2 shows the scatter plot for the objectives. The Pareto frontier is marked in red. Plot 1,3 shows that increasing gain is good for the first objective and plot 2,3 shows that decreasing gain is good for the second objective. The plots of the forth column shows that Td needs to be above 3. This is consistent with the well-known fact that the derivative term is needed for stabilization when controlling a double integrator with a PD controller.

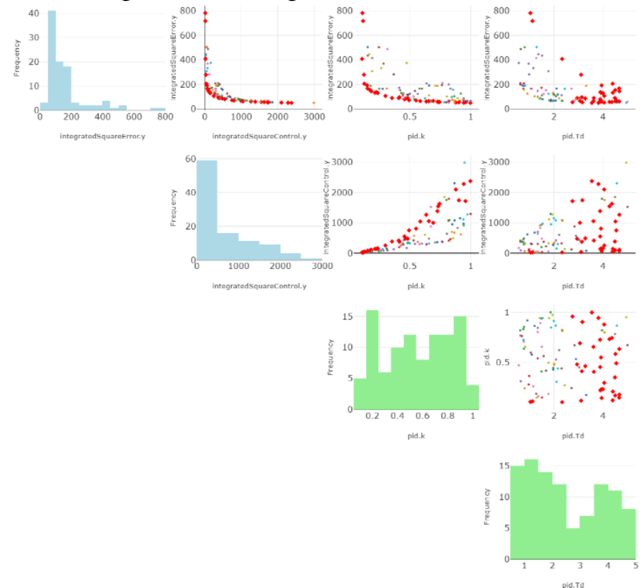


Figure 7. Multi-plot matrix with histogram and scatter plots.

By hovering over the plots, the actual values of objectives and tuners are presented:

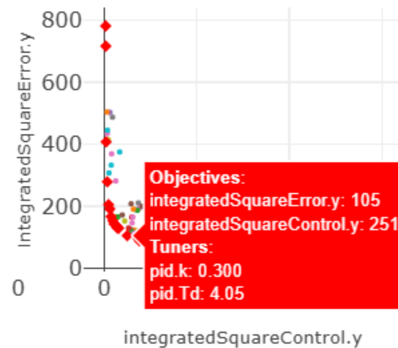


Figure 8. Tool tips showing objectives and tuners.

By zooming in on the Pareto frontier, the corresponding runs are plotted:

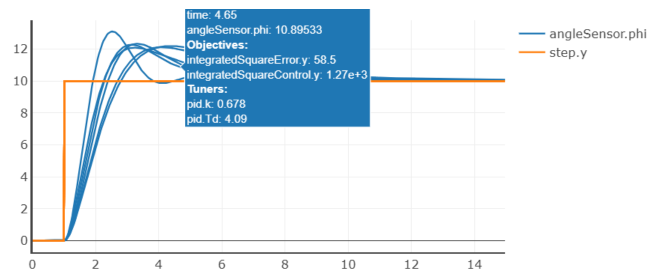


Figure 9. Selected responses.

If uncertainties are defined, for example, as a range of loads:

```
inertia1(J=uniform(1, 0.1, 1))
```

the plots include error bars and the tooltip shows the variations:

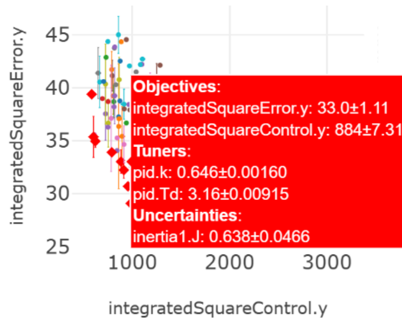


Figure 10. Scatter plot with error bars for uncertainties.

The error bars are obtained by defining bins for the tuner values and for each bin, the mean and standard deviation are presented. For accurate analysis of uncertainty influences many simulations are needed. The use of serverless functions on the cloud fits this use case.

8 Separate Translation of Models

The object-oriented modeling approach of Modelica allows building large models with millions of equations. However, the semantic specification is based on flattening, i.e., recursive cloning of variables and equations of each component instance. Typically, Modelica tools also expand matrix equations to scalar equations. Flattening means that a lot of memory is needed for variables and equations during translation of the Modelica code. It also means that translation time is long since the same analysis (flattening, symbolic processing, etc.) is performed repeatedly for each instance. Furthermore, the C-code becomes large, and compilation takes a long time. In addition, the execution of the C-code gets slower since the code might be larger than the size of the code cache.

However, parts of the equations of a component are always executed in the same order and with the same causality independently of how the component is connected. Such sequences of equations can be put into functions: less code gives shorter compilation time; less machine code gives shorter simulation time.

Finding such sequences can be made once for each model class. The technique is based on forming the most general environment to a model class as illustrated in **Figure 11**.

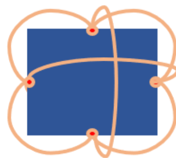


Figure 11. Generic environment of a model.

The generic environment is constructed by adding a set of dummy equations, env, which all depend on all connector variables (c_p: potential variables, c_f: flow variables) All inputs u are assumed to be known. The number of equations of env is the number of flow variables.

The Block Lower Triangular (BLT) form obtained during structural analysis of the model plus the generic environment has the structure shown in **Figure 12**.

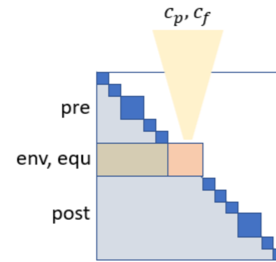


Figure 12. BLT structure of model plus generic environment.

The block in the middle contains the env equations since they all have the same incidences. It also contains certain equations, equ, which are related to the variables of the environment.

An example is given in the Appendix which shows how the equations are partitioned for a second order low pass filter.

9 Technology and Development Environment

The implementation of Modiator relies on *ECMAScript* modules[17]. Type inference and checking is done by *Visual Studio Code* [18] with *TypeScript* engine. Module dependency diagrams are generated by a *Visual Studio Code* plug-in, *Dependency Cruiser*.

The Modiator parser for Modelica is generated with the PEG-parser generator Peggy[19]. The handling of extends and merging of modifiers is done by deep merging of Javascript objects in a way inspired from merging in Modia.

Modelica model diagrams and 3D animations are created with three.js[20]. Plotting is done with Plotly[21].

Alias handling, tearing, index reduction, state-selection, BLT, symbolic processing of expressions, etc. have been ported from Modia (Julia code) to Javascript and some parts significantly improved. This has in some cases been done with a semi-automatic process using AI techniques with tool *Code Converter*[22] and the ChatGPT-4 tools *Microsoft Copilot*[23] and *Perplexity*[24]. Overall, the automatic translation reduced significantly the work to transform Modia functions to Javascript functions.

[17] https://nodejs.org/api/esm.html
[18] https://code.visualstudio.com/
[19] https://peggyjs.org/
[20] https://threejs.org/
[21] https://plotly.com/
[22] https://codeconverter.com/
[23] https://copilot.microsoft.com
[24] https://www.perplexity.ai/

DOI 10.3384/ecp218211

In order to enable building a full Modelica compiler and simulator running in the browser, C-code must be handled since Modelica has an interface to C-functions. Successful experiments have been made to use clang running under Wasmer[25] in the browser to compile C-functions into WebAssembly.

Modeling and simulation workflows typically involve scripting, for example with Python. For this reason, successful experiments have been made to use PyScript[26] and Pyodide[27] which enables running Python in the browser together with Modiator.

10 Stream with Media Propagation

When simulating fluid systems, there is currently the inherent drawback in Modelica that a medium has to be defined *at every component* and that every change of the Medium requires *recompilation* of the model. A prototype Modelica package called *Stream* does no longer has these drawbacks.

In this new approach, all media are defined with *external functions* and the internal memory of the functions by *ExternalObjects*[28] (so pointers to memory allocated inside the functions). An ExternalObject is constructed at one component and then propagated through connections. Below, first the principle of the approach is explained based on media *C-functions* and afterwards it is sketched how media *Javascript-functions* are supported in current Modiator.

Stream is a small subset of Modelica package *ThermofluidStream*[29] (Zimmer 2020, Zimmer et al. 2022) but with (a) different handling of the media inspired by (Otter et al. 2019) and (b) equations of the thermodynamic state formulated with *Linear Implicit Equilibrium Dynamics* (LIED) as proposed by Zimmer (2024, 2025). The latter means that all states are fixed in the library and no nonlinear equations appear in Modelica models due to Stream (but might be present inside the external functions). Separate translation, as proposed in section 8, becomes particularly easy in this case. A simple example of a system with boundaries, flow resistances and a volume are shown in the next figure.

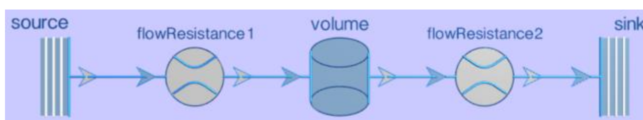


Figure 13. Simple Stream model in Modiator. The *medium* and its variables are defined in the *source* and propagated through the connections. The medium variables are changed in every component. *Media states* are defined in the *volume*. *Mass flow rate states* are defined in the *source* and in the *volume*.

The core LIED equations of the volume are shown in the next Listing.

Listing 4. Stream model of a volume with fixed size.

```
model Volume "Volume of fixed size"
  Interfaces.Inlet  inlet; Interfaces.Outlet outlet;
  parameter SI.Volume V_par "Fixed volume";
   ...
  Real xs[2](each fixed=false, each unit="1")
                                           "Scaled pair (p,T), (p,h), ...";
  Real Xi[size(Xi_start,1)](start=Xi_start, each fixed=true,
                       each unit="1") "Independent mass fractions";
protected
  parameter Integer mediumID(fixed=false);
  functions.Media.MediumAndState obj =
                      functions.Media.MediumAndState(mediumID);
  Real obj_r "Dummy, for correct sorting of obj";
initial equation
  mediumID = functions.Media.mediumID_from_obj(inlet.obj);
  xs         = functions.Media.get_xs(obj,p_start,T_start,...);
equation
  // Update obj with xs, Xi
  obj_r = functions.Media.updateThermodynamicState(obj, xs, Xi);
  outlet.obj = obj;  outlet.obj_r = obj_r;

  // Mass balance
  der_M = inlet.m_flow - outlet.m_flow ;
  der_d = der_M/V_par;

  // Energy balance
  d_in  = functions.Media.density(inlet.obj, inlet.obj_r );
  h_in  = functions.Media.specificEnthalpy(inlet.obj, inlet.obj_r );
  u_out = functions.Media.specificInternalEnergy(
                                          inlet.obj,inlet.obj_r );
  d_out = functions.Media.density(obj, obj_r);
  h_out = functions.Media.specificEnthalpy(obj, obj_r);
  M     = V_par*d_out;
  der_u = (inlet.m_flow*h_in - outlet.m_flow*h_out -
                                               der_M*u_out)/M;
  der(xs) = functions.Media.get_der_xs(obj, obj_r,der_d, der_u);

  // Mass fraction balance
  Xi_in  = functions.Media.massFractions(obj, obj_r);
  der(Xi) = (Xi_in*inlet.m_flow - Xi*(outlet.m_flow - der_M))/M;

  // Mass flow rate and inertial pressure (as in ThermofluidStream)
  ...
end Volume;
```

The state vector of a single substance compressible fluid medium is defined by two intensive quantities. In Modelica.Media, the state pairs (p,T), (p,h), (p,s), (d,T) are used. The pair selected for a particular medium in Stream is *hidden* and is implicitly defined by the medium name. In **Listing 4**, vector xs[2] is defined as a scaled pair, so that the two elements have magnitudes around 1 which means no nominal values need to be set for xs. For example, Modelica.Media.Air.SimpleAir uses the scaling $xs[1] = p/10^5$, $xs[2] = T/273.15$.

For a *multi-substance medium*, additionally the independent mass fractions Xi[:] are used as states. The dimension of Xi is identical to the dimension of its start value vector, i.e., `Real Xi[size(Xi_start,1)]`.

[25] https://wasmer.io/
[26] https://pyscript.net/
[27] https://pyodide.org/
[28] https://specification.modelica.org/maint/3.6/functions.html, section 12.9.7
[29] https://github.com/DLR-SR/ThermofluidStream

The internal medium states and the internal medium equations are defined by the declaration of the ExternalObject:

```
MediumAndState obj = MediumAndState(mediumID);
```

The input argument to the constructor is the Integer parameter mediumID – a unique identification of a Medium. It is deduced from the inlet input variable obj that references the medium and the medium variables in the connector:

```
parameter Integer mediumID(fixed=false);
initial equation
  mediumID = functions.Media.mediumID_from_obj(inlet.obj);
```

Since inlet.obj is formally a time varying variable propagated through connectors, the mediumID is determined during initialization via the given initial equation. This is not strictly Modelica compliant because alias equations for an ExternalObject are present (such as outlet.obj = obj). This tiny issue is not detected by Dymola[30] and by Modiator. The implementation of the ExternalObject constructor is (media.c contains the C-functions of the media):

```
class MediumAndState
  extends ExternalObject;
  function constructor
    input Integer mediumID "ID of medium";
    output MediumAndState obj;
    external "C" obj = MediumAndState_constructor(mediumID)
      annotation(IncludeDirectory=
                    "modelica://Stream/Resources/C-Sources",
          Include="#include \"media.c\"");
  end constructor;
```

Since der(xs) and der(Xi) appear in the volume model, xs and Xi are treated as known. With statement

```
obj_r = functions.Media.updateThermodynamicState(obj, xs, Xi);
```

the states xs and Xi are copied into obj. The function returns the Real dummy variable obj_r (short for obj_ready). Whenever, an access function is called with obj, obj_r must be provided as well to guarantee that obj is used after it was updated. The derivative of xs – der(xs) – is computed with the external function call

```
der(xs) = get_der_xs(obj, obj_r, der_d, der_u);
```

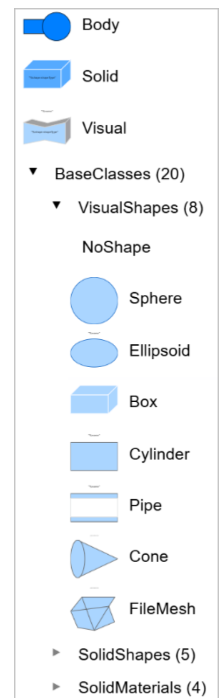
which has obj, obj_r, der_d (derivative of density) and der_u (derivative of specific internal energy) as arguments.

Modiator can currently not utilize C-functions. Therefore, Javascript functions that are equivalent to the functions in media.c are provided in file media-functions.js. This file is included whenever Modiator is started. During parsing of a Modelica model, all function names that start with “functions.” are interpreted to be Javascript functions and the look-up is not done in the Modelica model but internally in Modiator.

11 Model3D with 3D Geometries

The prototype Modelica package *Model3D* enables incorporation of 3D geometries into Modelica models. Currently, this capability is applied only to 3D mechanical systems but is intended to extend to other domains in the future. The design of Model3D is inspired by Modia3D[31] (Neumayr and Otter 2018, Neumayr 2025) and Modelica.Mechanics.MultiBody[32] (Otter et al. 2003). It utilizes the new approach “Dialectic Mechanics” introduced in (Zimmer and Oldemeyer 2023) and further developed in (Zimmer 2024, 2025). In particular, the states of a multibody system are fixed in the library and are the generalized joint coordinates of the spanning tree. Kinematic loops can be optionally modelled with elastic cut-joints. By introducing “kinetic velocities” which are filtered position derivatives, very stiff elastic joints can be reasonably simulated. The *analytic* handling of many kinematic loop types supported by the MultiBody library, will be ported to Model3D.

Primitive geometries, such as box, sphere etc. as well as meshes (.stl, .obj/mtl, .dae, .glb format) can be used in a model both for visualization and to compute properties such as mass and inertia, similarly as it is done in Modia3D. A screenshot from Modiators parameter menu of the basic parts of Model3D are shown at the right side. BaseClasses are replaceable models that can be utilized in Body and Visual (VisualShapes) as well as in Solid (SolidShapes, SolidMaterials). An example of a double pendulum with object diagram and animation is shown in **Figure 14**.



Model3D stores layout and settings parameters no longer in the World and other models, but in an external hierarchical JSON file. Standard Modelica tools read this file and extract content via Modelica package *ExternData*[33] (Beutlich and Winkler 2021).

Modiator accesses the settings JSON file directly from Javascript. In order that both approaches work seamlessly together, a few access routines of ExternData are replicated in Model3D as functions.getJSONxxx. If the Modiator parser recognizes function names starting with “functions”, it utilizes its Javascript implementation.

[30] https://www.3ds.com/products/catia/dymola

[31] https://github.com/ModiaSim/Modia3D.jl

[32] https://doc.modelica.org/Modelica%204.0.0/Resources/helpDymola/Modelica_Mechanics_MultiBody.html

[33] https://github.com/modelica-3rdparty/ExternData

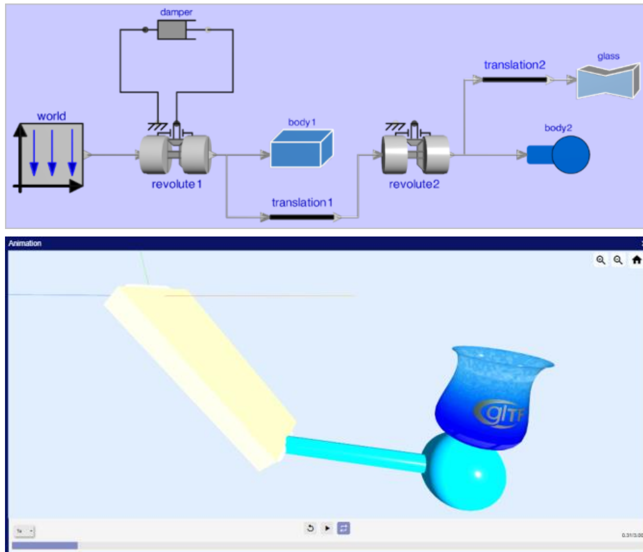


Figure 14. Model of a double pendulum in the diagram and animation widget of Modiator: *body1* is a "Solid" (here: Box) where mass properties are computed from the geometry and from the selected material, *body2* is a "Body" defined by mass, inertia and center of mass together with a "Visual" object (default: Cylinder) and optional marking of the center of mass with a sphere; *glass* is a "Visual" object (here: FileMesh).

As an example, part of the layout parameterization of a Revolute joint is shown in the next Listing.

Listing 5. Layout parametrization of Revolute joint.

```
constant String resource = Modelica.Utilities.Files.loadResource(
                  "modelica://Model3D/Resources/settings.json");
constant Real length = functions.getJSONReal(resource,
                                  "shape_in_m.jointLength",0.1);
constant Real diameter = functions.getJSONReal(resource,
                                  "shape_in_m.jointWidth",0.05);
constant Integer color[3] = functions.getJSONIntegerArray1D(
                        resource,"color.Revolute",3,{255,0,0});
 …
// settings.json file
 {"shape_in_m" : {
      "jointLength": 0.1,
      "jointWidth" : 0.05,
      "forceLength": 0.1,
      "forceWidth" : 0.05,
      "bodyDiameter": 0.11},
   "color": {
      "Revolute" : [255,0,0],
       …}
```

The functions.getJSONxxx act as a thin layer over the ExternData API. They have an additional default argument which is used if the settings file is unavailable. This technique results in much cleaner menus. Notably, all the many parameters of the World object are removed, except for the gravity definition which remains in the World object and is propagated via the connections. Consequently, global data is no longer present, so the inner prefix of World is removed.

12 Limitations and Future Work

The status of Modiator can be categorized as a Proof-Of-Concept for a complete Modelica compiler and simulator running in a web browser. So far, the focus has been to investigate an entire tool chain for the fundamental semantics of Modelica: model classes, component declarations with modifiers and redeclare, and connections.

It is sometimes assumed that the Modelica model is correct since many checks are not performed in the interest of translation speed. Scalability has not been in focus yet, i.e. only models with less than 1000 states and 5000 variables have been tested. The translator can be optimized in many ways, for example, by introducing caching and the separate translation technique outlined in section 8.

Only models which don't have nonlinear algebraic equation systems or mixed simultaneous equations with Real and Boolean unknowns can currently be simulated. Only very limited event handling is currently available. Other features not supported yet are:

- Modelica functions and external C-functions
- inner/outer
- complete set of matrix expressions
- algorithms
- initial equations/algorithms
- stream connectors (not planned)
- overconstrained connections (not planned)
- …

There are no plans to support the following parts of the Modelica Standard Library because they shall be replaced by better approaches:

(1) Modelica.Media shall be replaced by the Stream library sketched in section 10.
(2) Modelica.Fluid shall not be supported but instead the Stream library shall be used, and/or a future version of the ThermofluidStream library based on the Stream media handling and on LIED equations.
(3) Modelica.Mechanics.MultiBody shall be replaced by Model3D. Potentially, a nearly MultiBody compliant library will be provided as thin layer over Model3D to allow reasonable conversion of the many libraries and models that are based on the MultiBody library.

Since the current version of Modiator already (a) is useable on any device, (b) can be utilized just with the Modiator URL within a few seconds without installation, and (c) can translate and simulate smaller Modelica models locally in the browser nearly instantaneously, Modiator is well suited for *tutorials* and *university courses* on Modelica and for dedicated *web apps with a special user-oriented Web GUI* where the Modiator engine is running in the background.

13 Conclusions

Modiator is the next phase of experimentation and evolution of object-oriented modelling conducted by the authors after the Modia project. New web technologies provide the means to enable quick and easy access to object-oriented modeling and cloud simulation by Modelica Instant Simulation.

Furthermore, Modiator provides a platform for further investigations of new modeling capabilities to meet more and more demanding modelling needs.

Acknowledgements

The authors want to thank Johan Furuhjelm (Lund Institute of Technology) for the implementation of the infinite canvas user interface, Andreas Pfeiffer (DLR-FK) for valuable help regarding WebAssembly and Sundials and Dirk Zimmer (DLR-RM) for useful discussions regarding LIED and dialectic mechanics.

References

Bezanson, Jeff, Alan Edelman, Stefan Karpinski and Viral B. Shah (2017). “Julia: A fresh approach to numerical computing”. In: *SIAM review* 59.1, pp. 65–98. DOI: 10.1137/141000671

Beutlich, Thomas and Dietmar Winkler (2021): „Efficient Parameterization of Modelica Models”. In: Proceedings of the *14th International Modelica Conference.* DOI: 10.3384/ecp21181141

Elmqvist, Hilding, Alexander D. Baldwin and Simon Dahlberg (2015): “3D Schematics of Modelica Models and Gamification”. In: Proceedings of *11th International Modelica Conference*, pp. 527–536. DOI: 10.3384/ecp15118527

Elmqvist, Hilding, Martin Malmheden and Johan Andreasson (2018). “A Web Architecture for Modeling and Simulation”, In: Proceedings of the 2nd Japanese Modelica Conference, Tokyo, Japan, May 17-18, 2018

Elmqvist, Hilding, Martin Otter, Andrea Neumayr and Gerhard Hippmann (2021). “Modia - Equation Based Modeling and Domain Specific Algorithms”. In: Proceedings of *14th International Modelica Conference*, pp. 73–86. DOI: 10.3384/ecp2118173.

Franke, Rüdiger (2014): “Client-side Modelica powered by Python or Javascript”. In: *Proceedings of 10th International Modelica Conference*, pp. 1105-1112. DOI: 10.3384/ecp140961105

Gardner, David J. et al. (2022): “Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers”. In: *ACM Transactions on Mathematical Software (TOMS)*. 48.3, pp. 1-24, DOI: 10.1145/3539801

Hindmarsh, Alan C. et. al. (2005): “SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers”. In: *ACM Transactions on Mathematical Software (TOMS)*, 31.3, pp. 363–396. DOI: 10.1145/1089014.1089020

Kulhánek Tomáš, Arnošt Mládek, Filip Ježek and Jirí Kofránek (2023): “Bodylight.js 2.0 - Web components for FMU simulation, visualisation and animation in standard web browser”. In: *Proceedings of 15th International Modelica Conference*, pp. 443-451. DOI: 10.3384/ecp204443

Modelica Association (2023). *Modelica – A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.6*. URL: https://specification.modelica.org/maint/3.6/MLS.html.

Neumayr, Andrea and Martin Otter (2018): “Component-Based 3D Modeling of Dynamic Systems”. In: Proceedings of the *1st American Modelica Conference*. DOI: 10.3384/ECP18154175

Neumayr, Andrea (2025): *Modeling and Simulation of Physical Systems with Variable Structure*. Doctoral Dissertation, Technical University of Munich.

Otter, Martin, Hilding Elmqvist and Sven Erik Mattsson (2003): “The New Modelica MultiBody Library”. In: Proceedings of the 3rd International Modelica Conference. URL: http://www.Modelica.org/Conference2003/papers.shtml

Otter, Martin, Hilding Elmqvist, Dirk Zimmer and Christopher Laughman (2019), “Thermodynamic Property and Fluid Modeling with Modern Programming Language Constructs”. In: Proceedings of the *13th International Modelica Conference*, pp. 589–598. DOI: 10.3384/ecp19157589.

Otter, Martin and Hilding Elmqvist (2025), “Resizable Arrays in Object-Oriented Modeling”, In: Proceedings of the *16th International Modelica and FMI Conference.*

Short, Tom (2014): *OpenModelica models in Javascript*. URL: https://github.com/tshort/openmodelica-javascript

Zimmer, Dirk (2020): “Robust object-oriented formulation of directed thermofluid stream networks”. In: *Mathematical and Computer Modelling of Dynamical Systems*”, Taylor & Francis. DOI: 10.1080/13873954.2020.1757726

Zimmer, Dirk, Michael Meißner and Niels Weber (2022): The DLR ThermoFluid Stream Library. Electronics, Vol. 11, nr. 22, DOI: 2079-9292/11/22/3790

Zimmer, Dirk and Carsten Oldemeyer (2023): “Introducing Dialectic Mechanics”. In: Proceedings of the *15th International Modelica Conference 2023*, pp. 167–176. DOI: 10.3384/ecp204167.

Zimmer, Dirk (2024): “Object-Oriented Implementation of a Simulator for Linear Implicit Equilibrium Dynamics”. In: *ASIM 2024 Tagungsband*. DOI: 10.11128/arep.47.a4735.

Zimmer, Dirk (2025): “The Value of Enforcing a Strict Modeling Methodology within Modelica”. In: Proceedings of the *16th International Modelica & FMI Conference.*

Appendix
Example for Separate Translation

This appendix shows an example of separate translation of models as sketched in section 8. Consider the second order low pass filter[34] in **Figure 15**

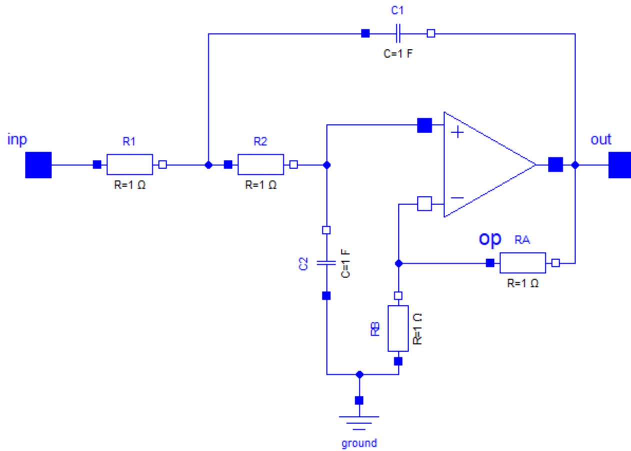


Figure 15. Second order low pass filter.

A corresponding Modelica model is shown below.

Listing 6. Modelica model of **Figure 15**.

```
model LowPass2nd
  Modelica.Electrical.Analog.Basic.Resistor R1(R=1);
  Modelica.Electrical.Analog.Basic.Resistor R2(R=1);
  Modelica.Electrical.Analog.Ideal.IdealOpAmp3Pin op;
  Modelica.Electrical.Analog.Basic.Ground ground;
  Modelica.Electrical.Analog.Basic.Capacitor C1(C=1);
  Modelica.Electrical.Analog.Basic.Capacitor C2(C=1);
  Modelica.Electrical.Analog.Basic.Resistor RA(R=1);
  Modelica.Electrical.Analog.Basic.Resistor RB(R=1);
  Modelica.Electrical.Analog.Interfaces.Pin inp;
  Modelica.Electrical.Analog.Interfaces.Pin out;
equation
  connect(R1.n, R2.p);
  connect(R2.n, C2.n);
  connect(ground.p, C2.p);
  connect(op.out, RA.n);
  connect(ground.p, RB.p);
  connect(R1.n, C1.p);
  connect(C1.n, op.out);
  connect(R2.n, op.in_p);
  connect(RB.n, op.in_n);
  connect(RA.p, RB.n);
  connect(R1.p, inp);
  connect(op.out, out);
end LowPass2nd;
```

If a voltage source is connected between ground and connector inp, the translated model with alias elimination applied will consist of a sequence of 10 solved equations.

If the filter is separately translated according to the algorithm outlined in section 8, an equivalent Modelica model is created (parameter handling simplified):

Listing 7. Modelica model of **Listing 6** transformed into a mix of acausal equations and causal functions.

```
model LowPass2nd
  Modelica.Electrical.Analog.Interfaces.Pin inp;
  Modelica.Electrical.Analog.Interfaces.Pin out;
equation
  LowPass2nd_init = ast.newFunction("C1_v, C2_v",
   "return {
     RA: {R: 1, p:{}, n:{}},
     RB: {R: 1, p:{}, n:{}},
     R1: {R: 1, p:{}, n:{}},
     R2: {R: 1, p:{}, n:{}},
     C1: {C: 1, v: C1_v, p:{}, n:{}},
     C2: {C: 1, v: C2_v, p:{}, n:{}},
     op: {in_p: {}, in_n: {}, out: {}},
     ground:{p:{}},
     out: {}}")

  LowPass2nd_pre = ast.newFunction("M", "
     M.RB.i = M.C2.v / M.RB.R;
     M.RA.v = M.RA.R * (-M.RB.i);
     M.RA.n.v = -((M.RA.v - M.C2.v));
     M.C1.p.v = M.C1.v - (-M.RA.n.v);
     return [M.RA.n.v, M.C1.p.v];");

  LowPass2nd_post = ast.newFunction("M, R1_p_i", "
     M.R2.v = M.C1.p.v - M.C2.v;
     M.R2.i = M.R2.v / M.R2.R;
     M.C1.p.i = -((-R1_p_i + M.R2.i));
     M.C1.der_v = M.C1.p.i / M.C1.C;
     M.C2.der_v = M.R2.i / M.C2.C;
     return [M.C1.der_v, M.C2.der_v];");

     M = LowPass2nd_init(C1_v, C2_v);
     (out.v, R1_n_v) = LowPass2nd_pre(M);

     R1_R = 1;
     R1_R * inp.i = R1_v;
     R1_v = inp.v - R1_n_v;

     (der(C1_v), der(C2_v)) = LowPass2nd_post(M, inp.i);
end LowPass2nd;
```

The equations are then partitioned into

- function LowPass2nd_pre with 4 assignments,
- function LowPass2nd_post with 5 assignments,
- and 2 remaining model equations.

If many instances of LowPass2nd are created, the 9 equations of the functions LowPass2nd_pre and LowPass2nd_post need to be compiled one time.

The functions LowPass2nd_... were defined using a built-in function ast.newFunction with the same semantics as the Javascript constructor Function().

[34] https://electronics.stackexchange.com/questions/271857/second-order-low-pass-filter-configuration